Assignment 1: Mutation Adequacy

Shana Chan (301543848) Michelle Vong (301421211) Kenneth Tan (301553450)
Aryan Rashid (301443517)

CMPT 473 E100
Rob Cameron
January 27, 2026

1 Overview

This assignment evaluated mutation adequacy on four Python modules selected from `TheAlgorithms/Python`. Each group member generated and executed at least 100 mutants for a chosen source file using an automated mutation tool (Cosmic Ray or Mutmut), then used the module's localized tests (pytest or doctest-derived tests) to classify each mutant as `KILLED` or `SURVIVED`. Surviving mutants were further analyzed to distinguish (i) **equivalent mutants** and (ii) **non-equivalent survivors** that indicate missing assertions, insufficient case coverage, or untested paths.

The primary metric reported is **test effectiveness**, defined as the percentage of **non-equivalent mutants killed**. Results varied by file: Jaro–Winkler achieved very high effectiveness due to strong numeric oracles for similarity scores, while the red-black tree file exhibited many surviving mutants in deletion repair and invariant-checking logic. Across all modules, mutation testing surfaced gaps that were not obvious from correctness checks alone, especially when tests validated final outputs but not *internal invariants* (e.g., tree coloring rules, list tail pointers, or matrix iteration bounds). In several cases, a *single* additional invariant assertion would have killed multiple survivors, suggesting that the most cost-effective improvements are often *broad* assertions rather than many narrowly tailored tests.

2 Project Proposal

2.1 Identification of the Open-Source Project

We analyzed selected modules from `TheAlgorithms/Python`, an open-source repository of algorithm implementations written in Python: https://github.com/TheAlgorithms/Python. The modules studied were `circular_linked_list.py`, `red_black_tree.py`, `jaro_winkler.py`, and `haralick_descriptors.py`.

2.2 Supporting Organization

The project is developed and maintained by the open-source TheAlgorithms community, with contributions from a large developer base. The project also accepts financial support through platforms such as Liberapay: https://liberapay.com/TheAlgorithms. For this study, we treated the repository as a representative, actively maintained Python codebase with a broad user and contributor base.

2.3 Size of the Code Base

The repository contains approximately 1.38 million lines of code as measured by `cloc` (excluding comments and blank lines).

2.4 Test Suite Existence

While the repository does not ship a single monolithic test suite, it contains substantial module-level tests and doctests; we evaluated the test suite(s) associated with each selected module.

2.5 Evaluation Platform (OS, Language)

Our team will perform mutation testing using Cosmic Ray and Mutmut in a Docker container with an Ubuntu-based image, ensuring the same operating system, Python version, and dependencies are used when evaluating the test suites.

2.6 Build Time

This project is written in Python, so no compilation or linking step is required. The setup involves installing dependencies via ‘pip‘ within the Docker container, which takes only a few seconds to complete.

2.7 Test Execution Time

The repository does not provide a single global test suite. Instead, individual modules include their own localized tests. Test execution times are therefore reported separately for each source file.

Table 1: Measured module-level test execution times in our evaluation environment.

Member	Module	Test Suite Time
Michelle Vong	`circular_linked_list.py`	0.83 ms
Shana Chan	`red_black_tree.py`	0.56 ms
Kenneth Tan	`haralick_descriptors.py`	≈0.016 s per run
Aryan Rashid	`jaro_winkler.py`	≈2–3 s

3 Mutation Adequacy Study

3.1 Effectiveness Calculation

Mutants found to be equivalent were excluded from effectiveness calculations. To calculate effectiveness, the formula below will be used. Survived mutants are those not killed by the test suite; this set includes equivalent mutants. The formula will be used both at the routine level (using counts specific to each routine) and at the file level (using totals aggregated across all routines in the file).

$$\text{Effectiveness} = \frac{K}{M - E} \times 100\%.$$

3.2 Integrated Results Overview

Table 2: File-level mutation results and effectiveness (equivalent mutants excluded).

File (Member)	Total	Killed	Eq.	Eff.
`circular_linked_list.py` (Michelle)	184	150	3	82.9%
`red_black_tree.py` (Shana)	328	143	22	46.7%
`jaro_winkler.py` (Aryan)	273	257	2	94.8%
`haralick_descriptors.py` (Kenneth)	290	211	19	77.9%

Cross-file observations.

- *Value-oracle strength matters.* Modules with crisp numeric outputs (e.g., similarity scores, normalization outputs) can achieve high mutation effectiveness with a modest set of assertions.
- *Invariant testing is a recurring gap.* Data structure implementations often survived mutations that preserved outward behavior while breaking internal invariants (tail pointers, parent pointers, red-black properties).
- *Control-flow case coverage drives adequacy.* Deletion-repair logic in red-black trees exhibited many surviving mutants due to unexercised case branches.
- *Small fixtures beat large inputs.* For image/matrix logic, tiny hand-checkable inputs reliably killed off-by-one/bounds mutants that larger inputs can accidentally mask.

3.3 Michelle Vong: `circular_linked_list.py` (Cosmic Ray)

Mutation operators used: AOR, ROR, LCR, CRP/SRC, UOI, SDL, RSR.

Methods and Results. Cosmic Ray was used to apply mutation operators to `circular_linked_list.py` and execute tests for each mutant. The routines of primary interest were `__iter__`, `delete_nth`, and `insert_nth`, because pointer-updating logic is a common source of subtle faults in linked structures. The lowest routine-level effectiveness occurred in `delete_nth`, where multiple surviving mutants broke tail updates without affecting simple head-based iteration outcomes. This suggests the tests are primarily *sequence*-oriented (values produced by iteration) rather than *structure*-oriented (pointer relationships that must hold after edits).

Table 3: Routine-level mutation results for `circular_linked_list.py`.

Routine	Tot.	Kill.	Eq.	Eff.	Routine	Tot.	Kill.	Eq.	Eff.
`__iter__`	10	9	1	100.0%	`insert_head`	2	2	0	100.0%
`delete_nth`	80	55	0	68.8%	`delete_front`	2	2	0	100.0%
`insert_nth`	66	61	1	93.8%	`delete_tail`	15	13	0	86.7%
`__len__`	2	2	0	100.0%	`is_empty`	7	6	1	100.0%
`__repr__`	0	0	0	N/A	**File total**	184	150	3	82.9%
`insert_tail`	0	0	0	N/A					

Table 4: Five representative survivors for `circular_linked_list.py`.

Routine	Class.	Rationale / test direction
`__iter__`	Eq.	`node == head` vs `node is head` is equivalent for valid circular list states because traversal returns to the same head object; both become true at the same iteration step.
`delete_nth`	Non-eq.	Add a test that deletes the last index and asserts `tail.data` and `tail.next_node == head`. The mutation disables the tail-update condition, so the tail pointer is never updated.
`delete_nth`	Non-eq.	Same tail-update failure mode as above via a different operator; killed by the same tail-pointer invariant test (above).
`insert_nth`	Non-eq.	Add tests that insert at the tail for multiple list sizes (e.g., lengths 2–6) and assert `tail.data` and `tail.next_node == head`. The mutation replaces `len(self)-1` with `len(self)^1`, altering when the tail is updated.
`insert_nth`	Eq.	`index==0` vs `index<=0` is equivalent because negative indices are rejected before this branch is reachable.

Observations and Improvements. Survivors were disproportionately related to tail pointer maintenance. The existing tests validated behavior by iterating from the head and checking values, which does not reliably expose broken tail invariants. The most efficient improvement is to add *invariant assertions* after insertions/deletions (tail correctness, tail-next equals head) in addition to output checks. This change would target multiple surviving mutants at once rather than adding many narrowly tailored cases. A second improvement is to add length-parameterized tests (e.g., sizes 1–6) to prevent accidental "passing" due to degenerate list sizes.

3.4 Shana Chan: `red_black_tree.py` (Mutmut)

Mutation operators used: Mutmut AST-level mutations (arithmetic, comparison, boolean, constant, string, keyword, and statement-level changes), plus operator families aligned with AOR/ROR/LCR/CRP/UOI/SDL/RSR where applicable.

Methods and Results. Mutmut was used to generate and execute mutants for `red_black_tree.py`. AST-level mutation is well-suited for this module because it avoids producing large numbers of trivial or invalid mutants, focusing instead on executable logic. Routine-level results showed strong adequacy for core structural operations (rotations, insert) but significantly weaker adequacy for deletion repair and property-checking routines. In particular, `_remove_repair` had low effectiveness, consistent with incomplete coverage of the routine's multiple case branches. Deletion scenarios are harder to cover exhaustively because they require constructing trees that trigger specific sibling/child color configurations.

Table 5: Routine-level mutation results for `red_black_tree.py`.

Routine	Tot.	Kill.	Eq.	Eff.	Routine	Tot.	Kill.	Eq.	Eff.
`rotate_left()`	12	11	0	91.7%	`check_color_properties()`	18	3	12	50.0%
`rotate_right()`	12	9	0	75.0%	`check_coloring()`	14	8	0	57.0%
`insert()`	23	17	0	73.9%	`black_height()`	17	7	0	41.2%
`_insert_repair()`	29	21	2	77.0%	`__contains__()`	2	2	0	100.0%
`remove()`	28	14	0	50.0%	`search()`	8	7	0	87.5%
`_remove_repair()`	108	18	8	18.0%	`floor()`	8	7	0	87.5%
`is_left()`	3	2	0	66.7%	`ceil()`	8	7	0	87.5%
`is_right()`	3	2	0	66.7%	`__bool__()`	1	1	0	100.0%
`__len__()`	6	0	0	0.0%	`__eq__()`	6	5	0	83.3%
`__repr__()`	20	0	0	0.0%	`color()`	2	2	0	100.0%
					File total	328	143	22	46.7%

Table 6: Five representative survivors for `red_black_tree.py`.

Routine	Class.	Rationale / test direction
`rotate_left`	Non-eq.	Add a test that performs a left rotation and asserts correct parent pointers for the new root, its left child, and any transferred subtrees. The mutation corrupts parent-pointer updates during rotation.
`_insert_repair`	Non-eq.	Add a black-uncle insertion case and assert correct recoloring of the parent and root after repair. The mutation assigns `None` to a node color, violating the intended $\{0, 1\}$ domain.
`remove`	Non-eq.	Add a test that deletes a node with two children and asserts that the value is removed while all other values remain searchable. The mutation disables the two-children deletion path.
`_remove_repair`	Non-eq.	Add test: construct a tree whose deletion triggers Case 3 (sibling black, sibling.left red, sibling.right black), then delete the target value and assert red-black invariants (root black, no double-red edges, and consistent black-height).
`check_color_properties`	Non-eq.	Add a test that constructs a deliberate double-red violation and asserts that `check_color_properties()` returns `False`. The mutation flips a failure return to `True`.

Observations and Improvements. The strongest tests checked structural properties such as ordering and parent-child relationships, which kills many rotation and insertion mutants. However, red-black invariants (no adjacent red nodes, consistent black-height, root black) were not asserted systematically after operations, allowing mutants that preserve BST shape but break color semantics to survive. The most valuable improvement is to add a compact invariant-check helper used by multiple tests, and to add targeted removal scenarios that exercise all `_remove_repair` case branches. In practice, even a minimal invariant helper (root black; no red-red edges; black-height equal on all paths) would substantially raise effectiveness.

3.5 Aryan Rashid: `jaro_winkler.py` (Cosmic Ray)

Mutation operators used: AOR, ROR, LCR, CRP/SRC, UOI, SDL, RSR.

Methods and Results. Mutants were generated for `jaro_winkler.py` using Cosmic Ray. The original tests were presented as doctests embedded with the implementation; for maintainability and compatibility with the mutation workflow, tests were rewritten as equivalent pytest assertions. Despite a small test suite, effectiveness was high because the algorithm has a clear numeric oracle: similarity scores can be checked directly. The primary weaker area was the helper routine that defines the matching window bounds, where subtle off-by-one changes can survive if tests do not include boundary-sensitive pairs. This module illustrates that when the oracle is precise, a small number of well-chosen test points can achieve high adequacy.

Table 7: Routine-level results for `jaro_winkler.py`.

Routine	Total	Killed	Eq.	Eff.
`jaro_winkler`	259	246	2	95.7%
`get_matching_characters`	14	11	0	78.6%
File total	273	257	2	94.8%

Table 8: Five representative survivors for `jaro_winkler.py`.

Area	Class.	Rationale / test direction
Window bounds	Non-eq.	Mutation shrinks the matching window, reducing similarity scores. Add a boundary-sensitive test pair with a known expected score that fails when the window is too small.
Prefix loop	Non-eq.	Replacing `==` with `is` relies on interpreter-dependent interning. Add a test that compares two distinct string objects with identical characters to enforce value equality rather than identity.
Prefix max	Non-eq.	Prefix comparison is capped at length 3 instead of 4. Add a test with two strings sharing a common prefix of length 4 and assert the expected Jaro–Winkler score.
Transpositions	Eq.	`//2` versus `/2` is equivalent because the transposition count is always even, so division always yields an integer result in valid executions.
Zero-match base	Eq.	`match_count` is never negative, so `== 0` and `<= 0` behave identically for all reachable states.

Observations and Improvements. Because the algorithm is mathematically defined, mutation analysis is especially interpretable: most surviving non-equivalent mutants correspond to real deviations from the standard procedure. The simplest improvement is broader value-based testing: include cases that isolate each major component (window bounds, prefix bonus length 0–4, and transpositions). A second improvement is to add a property-style test that enforces basic invariants (score in $[0, 1]$, symmetry for selected pairs if expected, and monotonic behavior under identical prefix extensions where applicable). Finally, adding a few near-duplicate string pairs (single swap; single insertion; repeated characters) would better constrain the matching-character logic.

3.6 Kenneth Tan: `haralick_descriptors.py` (Mutmut)

Mutation operators used: MOTHRA operator families (AOR, ROR, LCR, CRP) together with additional Mutmut AST-level mutations, including unary and return modifications (UOI, RSR), statement-level changes (SDL, SAN, DER, DSA, GLR), scalar and constant substitutions (SVR, CSR, SCR, SRC, ABS), and array/reference replacement operators (AAR, ACR, ASR, SAR, CAR, CNR).

Methods and Results. Mutation testing was performed with Mutmut in a Python 3.11 Docker environment with NumPy/ImageIO dependencies. Mutmut integrates with pytest and uses AST transformations to generate mutants systematically without manual edits. Numeric routines (normalization and distance) tended to have high effectiveness because their outputs are easily asserted. In contrast, some image/matrix routines had lower effectiveness because small changes to bounds or default control-flow paths are not always exposed by the existing tests. In several survivors, tests exercised the routine but did not validate *intermediate* results (e.g., exact co-occurrence counts), allowing deviations to persist.

Table 9: Routine-level mutation results for `haralick_descriptors.py`.

Routine	Tot.	Kill.	Eq.	Eff.	Routine	Tot.	Kill.	Eq.	Eff.
`root_mean_square_error()`	5	4	0	80.0%	`binary_mask()`	9	8	1	100.0%
`normalize_image()`	10	10	0	100.0%	`matrix_concurrency()`	38	15	0	39.5%
`normalize_array()`	12	10	2	100.0%	`haralick_descriptors()`	57	45	0	78.9%
`grayscale()`	10	9	1	100.0%	`get_descriptors()`	10	9	1	100.0%
`binarize()`	10	9	0	90.0%	`euclidean()`	5	4	0	80.0%
`transform()`	61	55	3	94.8%	`get_distances()`	22	17	0	77.3%
`opening_filter()`	20	9	11	100.0%	**File total**	290	211	19	77.9%
`closing_filter()`	21	7	0	33.3%					

Table 10: Five representative survivors for `haralick_descriptors.py`.

Routine	Class.	Rationale / test direction
`root_mean_square_error`	Non-eq.	Returning `None` violates the routine contract; add a test asserting the result is a float and matches a known RMSE on a small pair.
`normalize_array`	Non-eq.	Dividing by `cap` instead of multiplying changes scaling; add a non-default `cap` case and assert the full expected normalized array.
`grayscale`	Non-eq.	Including an alpha channel in the dot-product slice can cause a shape mismatch; add an RGBA fixture and assert alpha is ignored and grayscale equals the expected value.
`euclidean`	Non-eq.	Using addition instead of subtraction breaks the Euclidean distance definition; add asymmetric vectors with a known expected distance (e.g., $\sqrt{101}$).
`matrix_concurrency`	Non-eq.	Changing loop bounds alters co-occurrence counts; add a tiny hand-checkable image and assert the exact expected matrix.

Observations and Improvements. Mutation testing revealed a split: arithmetic-heavy helpers were well protected by output assertions, while routines involving iteration regions (filters, co-occurrence matrix) benefited most from tiny fixtures with exact expected intermediate outputs. The most effective upgrade is to introduce more *small, fully specified* test images where expected matrices can be derived by inspection, and to include boundary cases that constrain loop bounds and neighborhood definitions. A complementary improvement is to add type/shape assertions (e.g., output dimensions; dtype expectations) so that mutants causing silent broadcasting or shape drift are immediately detected.

4 Lessons Learned and Recommended Test Improvements

Across the four modules, the following improvements were consistently high value:

- *Assert invariants, not just outputs.* For data structures, test correctness should include internal pointer and invariant checks (tail correctness; red-black coloring rules; black-height consistency; parent pointers after rotation).
- *Design tests around case partitions.* Routines with multiple repair cases (e.g., red-black deletion) require tests that deliberately trigger each case rather than relying on incidental coverage.
- *Prefer tiny, exact fixtures for matrix/image logic.* Small hand-checkable inputs expose off-by-one and bounds mutations more reliably than large images, where errors may average out.
- *Treat equivalence explicitly.* Surviving mutants must be analyzed; equivalent mutants should be excluded so effectiveness reflects only meaningful test gaps.

5 Automation

Mutation generation and test execution in this study were already automated using Cosmic Ray and Mutmut. For each module, mutants were generated, tests were run, and outcomes were recorded without manual intervention; manual effort was limited to analyzing surviving mutants and classifying equivalence. These steps could be wrapped into a single project-level driver that iterates over all selected modules, producing a unified report for the entire study in one run.

The remaining manual steps could be automated with lightweight scripting. Mutation reports could be post-processed to compute routine-level effectiveness and generate summary tables directly. Surviving mutants could also be grouped automatically by routine or operator to guide focused analysis, making mutation adequacy suitable for repeated or CI-based use.